# EMBANKS: Towards Disk Based Algorithms For Keyword-Search In Structured Databases

Submitted in partial fulfillment of the requirements
for the degree of

**Bachelor of Technology**

**Nitin Gupta**

Advised by
**Prof. S. Sudarshan**

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

# Acknowledgement

I thank **Prof. S. Sudarshan** for his inspiration, invaluable guidance and support in making my B.Tech Project an informative and challenging experience for me. His enthusiasm and patience during the last one year have driven me all this while. I would also like to appreciate Prof. Soumen Chakrabarti for his inputs during all the examinations.

I thank my friends and labmates for bugging me at all the wrong times, but providing support when I needed it the most. A special thanks also to **Kumar Gaurav Bijay** for his valuable comments and suggestions throughout the project.

Nitin Gupta

## Abstract

In recent years, there has been a lot of interest in the field of keyword querying relational databases. A variety of systems such as DBXplorer [ACD02], Discover [HP02] and ObjectRank [BHP04] have been proposed. Another such system is BANKS, which enables data and schema browsing together with keyword-based search for relational databases. It models tuples as nodes in a graph, connected by links induced by foreign key and other relationships. The size of the database graph that BANKS uses is proportional to the sum of the number of nodes and edges in the graph. Systems such as SPIN, which search on Personal Information Networks and use BANKS as the backend, maintain a lot of information about the users' data. Since these systems run on the user workstation which have other demands of memory, such a heavy use of memory is unreasonable and if possible, should be avoided. In order to alleviate this problem, we introduce EMBANKS (acronym for External Memory BANKS), a framework for an optimized disk-based BANKS system. The complexity of this framework poses many questions, some of which we try to answer in this thesis. We demonstrate that the cluster representation proposed in EMBANKS enables in-memory processing of very large database graphs. We also present detailed experiments that show that EMBANKS can significantly reduce database load time and query execution times when compared to the original BANKS algorithms.

# Contents

# List of Figures

Chapter 1

# Introduction

In recent years, the field of keyword querying relational databases has received great attention. Database search engines have to cope with several challenges such as designing a representation for the database graphs that can efficiently support a diverse range of complex queries over various schema. Given the size and rapid improvement in disk-types, a relatively small database today has millions of tuples. To this end, prior and ongoing research projects such as DBXplorer [ACD02], Discover [HP02] and ObjectRank [BHP04] have been proposed. Given a set of query keywords, DBXplorer returns all rows (either from single tables, or by joining tables connected by foreign-key joins) such that the each row contains all keywords. ObjectRank [BHP04], on the other hand, is an authority-based keyword search engine for databases which returns a group of nodes containing all keywords.

With evolution of different techniques in this area of research, a uniform model has emerged for representing relational databases as a graph with the tuples in the database mapping to nodes and cross references (such as foreign key and other forms of references) between tuples mapping to edges connecting these nodes. Each tuple in the database is modeled as a node in the directed graph and each foreign key-primary key link as an edge between the corresponding tuples. This can be easily extended to other type of connections; for example, it can be extended to include edges corresponding to inclusion dependencies, where the values in the referencing column of the referencing table are contained in the referred column of the referred table but the referred column need not be a key of the referred table. Keywords in a given query activate some nodes. The answer to the query is defined to be a subgraph which connects the activated nodes.

BANKS [BNH$^+$02] (acronym for Browsing ANd Keyword Searching) is a system that enables data and schema browsing together with keyword-based search for relational databases. It models tuples as nodes in a graph, connected by links induced by foreign key and other relationships. Answers to a query are modeled as rooted trees connecting tuples that match individual keywords in the query. Recently, Kacholia et al [KPC$^+$05] proposed a bidirectional expansion algorithm for keyword search, an improvement over the original BANKS backward expanding search algorithm. The improvement has been suggested because the original BANKS algorithm performs poorly if some keywords match many nodes, or some node has very large degree. The size of the database graph that BANKS uses is proportional to the sum of the number of nodes and edges in the graph. Systems such as SPIN, which search on Personal Information Networks and use BANKS as the backend, maintain a lot of information about the users' data. Since these systems run on the user workstation which have other demands of memory, such a heavy use of memory is unreasonable and if possible, should be avoided.

In this thesis we introduce EMBANKS (acronym for External Memory BANKS), a framework for an optimized disk-based BANKS system. This framework is intended to support various search models (primarily BANKS) with data structures to facilitate external memory search, and thus alleviate the problem of excessive memory usage. EMBANKS, in a nutshell, proceeds as follows: it first clusters the given input graph based on various parameters and stores the graph thus obtained on disk. The search algorithm is then executed on this smaller graph to return a few answers, which are then expanded and the search algorithm is then re-executed on this expanded graph to get *real* answers. As described herein, EMBANKS, apart from reducing the required memory size, also speeds up the database load time and run time for various queries when compared to the both the existing algorithms. This basis of the framework leads us to a series of questions - which clustering techniques should be used? What should be the size of a single cluster? How will the various weights (for both nodes and edges) in a clustered graph be computed? What is the ideal number of *artificial* answers to generate? We try to provide answers to most of these questions in this thesis.

## 1.1 Problem Identification

| | DBLP | | IMDB | | IIT Movie | | US Patent | |
|---|---|---|---|---|---|---|---|---|
| | Num | Size | Num | Size | Num | Size | Num | Size |
| Nodes | 1.77M | | 1.74M | | 4.33K | | 2.23M | |
| nodeType | | 7.09 | | 6.96 | | 0.17 | | 8.94 |
| Prestige | | 7.09 | | 6.96 | | 0.17 | | 8.94 |
| adjacencyOffset | | 7.09 | | 6.96 | | 0.17 | | 8.94 |
| nodeIndeg | | 7.09 | | 6.96 | | 0.17 | | 8.94 |
| nodeOutdeg | | 7.09 | | 6.96 | | 0.17 | | 8.94 |
| Edges | 8.49M | | 7.94M | | 21.43K | | 11.88M | |
| AdjacentNode | | 33.99 | | 31.77 | | 0.86 | | 47.54 |
| Weight | | 33.99 | | 31.77 | | 0.86 | | 47.54 |
| Priority | | 33.99 | | 31.77 | | 0.86 | | 47.54 |
| Total | | **137.42** | | **130.11** | | **3.43** | | **187.32** |

Table 1.1: The minimum memory requirement (in MBytes) by BANKS

Table 1.1 lists the minimum memory requirement of BANKS at load time. Though this preprocessing and loading has to be done only once, the numbers show that this minimum requirement is significant even for moderately sized databases. The numbers presented in Figure 1.1 do not include the memory used by pointers and other such objects references which are essential for the execution. It also does not include the memory required for query execution variables such as random strings, priority queues and heaps.

Let us now consider a personal information network. Assuming 3 users, we estimate the number of files on the computer to be roughly half a million, with over five million-edges between them based on proximity and similarity. Given this, the minimum memory requirement of BANKS can be estimated to be $(0.5M \times 4 \times 5) + (5M \times 4 \times 3) = 70MB$!

## 1.2 Organization of the Thesis

This thesis has three main parts. In the first part, we explore various techniques that have been suggested for other systems that might be applicable for an external memory implementation of the BANKS algorithm. Clustering and multi-level traversal are two such techniques [RGM03].

Motivated by these systems, we develop EMBANKS through a series of several disk-based optimization techniques for the BANKS algorithm. We summarize the contributions of this part of the thesis in Chapter 3. In the third part of the thesis, we discuss accuracy and efficiency constraints for the EMBANKS framework and suggest solutions to improve the same. We summarize the contributions of this part of the thesis in Chapter 4. In Chapter 5, we present detailed experiments that show that EMBANKS can significantly reduce database load time and query execution times when compared to the original BANKS algorithms. We conclude the thesis with a discussion on the results obtained and possible future work, in Chapter 6.

Chapter 2

# Related Work

There has been a lot of interest recently in the field of keyword querying relational databases. A variety of systems such as DBXplorer [ACD02], Discover [HP02] and ObjectRank [BHP04] have been proposed. An interesting proposal for querying a web graph based on semantics has been proposed in SphereSearch [GSW05], which claims to support concept-aware, context-aware, and abstraction-aware search. We begin this chapter with a formal introduction to the graph model used by most of the database search engines today, followed by a detailed explanation of BANKS in Section 2.2, a brief look at the former systems in Section 2.3 and finally conclude in Section 2.4 with the main inspiration for this thesis - the web graph model.

## 2.1 An Introduction to the Graph Model

The formal graph model used by database search engines can be described as follows:

- Vertices: For each tuple $T$ in the database, the graph has a corresponding node $n(T)$. We will speak interchangeably of a tuple and the corresponding node in the graph.
- Edges: For each pair of tuples $T_1$ and $T_2$ such that there is a foreign key from $T_1$ to $T_2$, the graph contains an edge from $n(T_1)$ to $n(T_2)$ and a back edge from $n(T_2)$ to $n(T_1)$ (this can be extended to handle other types of connections).
- Edge weights: This weight assignment varies from one technique to another. Weight of a forward link along a foreign key relationship reflects the strength of the proximity relationship between two tuples and is normally set to 1 by default. It may be set to any desired value to reflect the importance of the link (low weights correspond to greater proximity). Let $s(R_1, R_2)$ be the similarity from relation $R_1$ to relation $R_2$ where $R_1$ is the referencing relation and $R_2$ is the referenced relation. Then the similarity $s(R_1, R_2)$ depends upon the type of the link from relation $R_1$ to relation $R_2$, and is different than the actual edge weights.
- Node weights: Each node $n$ in the graph is assigned a weight $W(n)$ which depends upon the prestige of the node. In simplest case it can be set to the in-degree of the node.

## 2.2 BANKS

This section describes BANKS, a system which enables keyword-based search on relational databases, together with data and schema browsing. BANKS enables users to extract information in a simple manner without any knowledge of the schema or any need for writing complex queries. A user can get information by typing a few keywords, following hyperlinks, and interacting with controls on the displayed results. Answers are ranked using a notion of proximity coupled with a notion of prestige of nodes based on inlinks, similar to techniques developed for web search.

The idea of proximity search in databases represented as graphs was also proposed by Goldman et al. They support queries of form find object near object. They restrict results to tuples from one relation near a set of keywords, whereas BANKS permits results to be structured as trees which helps explain how it arrived at an answer. Unlike BANKS, they do not consider node and edge weighting techniques.

### 2.2.1 Backward Expanding Search Model

The algorithm presented in BANKS [BNH$^+$02] models the database as a directed graph where each tuple is a node of the graph. Foreign-key-primary key links are modeled as directed edges between the corresponding tuples. The edges are directed since the strength of connections between two nodes is not necessarily symmetric. For example, consider the data graph of DBLP, which has a node called conference, connected to a node for each conference, which are then connected to papers published in those conferences. The path through the conference node is a relatively meaningless (compared to an authored-by edge from paper to author). This leads to a natural application of directed edges. The weight of such an edge along a foreign key relationship reflects the strength of the proximity relationship between two tuples. It can be set to any desired value to reflect the importance of the link (small values correspond to greater proximity).

Each answer tree is assigned a relevance score, and answers are presented in decreasing order of that score. The scoring described in the paper involves a combination of relevance clues from nodes and edges. Node weights and edge weights provide two separate measures of relevance. One of the desirables of the algorithm is to control the variation in individual weights so that a few nodes or edges with very large weights do not skew the results excessively.

Finding such a subgraph is a NP complete problem (computation of minimum Steiner trees). This is further complicated by node weight considerations, required to compute the overall relevance of a tree. The *backward expanding search algorithm* described in the paper offers a heuristic algorithm for incrementally computing query results. One of the initial assumptions of the algorithm is that the *graph fits in memory*.

The algorithm begins by looking up tuples containing the search keywords with the help of disk resident indices or symbol tables. Given a set of keywords, for each keyword term $t_i$ , there is corresponding set of nodes, $S_i$ , that are relevant to the keyword. All nodes belonging to any of these sets are marked and the main goal of the algorithm is to find a subgraph connecting these marked nodes. Since just by looking at the subgraph it is not apparent as to what information it conveys, the algorithm tries to identify a node in the graph as a central node that connects all keyword nodes, and also strongly reflects the relationship amongst them.

Let $S = \cup S_i$ . The backward expanding search algorithm concurrently runs $S$ copies of Dijkstra's single source shortest path algorithm, one for each keyword node $n$ in $S$, with $n$ as the source. The copies of the algorithm are run concurrently by creating an iterator interface to the shortest path algorithm, and creating an instance of the iterator for each keyword node. Each copy of the single source shortest path algorithm traverses the graph edges in reverse direction. Basically at each iteration of the algorithm, one of the iterators is picked for further expansion. The iterator picked is the one whose next vertex to be visited has the shortest path to the source vertex of the iterator (the distance measure can be extended to include node weights of the nodes matching keywords). A list of all the vertices visited is maintained for each iterator.

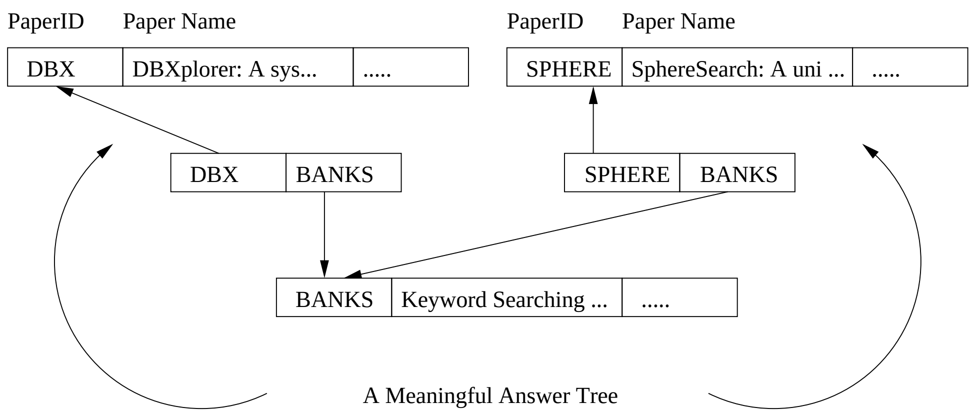


Figure 2.1: Example of an answer returned by BANKS.

Consider a set of iterators containing one iterator from each set $S_i$. If the intersection of their visited vertex lists is non-empty, then each vertex in the intersection defines a tree rooted at the vertex, with a path to at least one node from each set $S_i$. The idea is to find a common vertex from which a forward path exists to at least one node in each set $S_i$ . Such paths define a rooted directed tree with the common vertex as the root and the corresponding keyword nodes as the leaves. The tree thus formed will be a connection tree and root of the tree is the information node.

## 2.2.2 Bidirectional Search Model

In brief, backward expanding strategy described above does a best-first search from each node matching a keyword; whenever it finds a node that has been reached from each keyword, it outputs an answer tree. However, Backward expanding search may perform poorly w.r.t. both time and space in case a query keyword matches a very large number of nodes (e.g. if it matches a "metadata node" such as a table or column name in the original relational data), or if it encounters a node with a very large fan-in (e.g. the "paper appeared in conference" relation in DBLP leads to "conference" nodes with large degree). In other words, there are two scenarios in which backward search unnecessarily explores a large number of nodes:

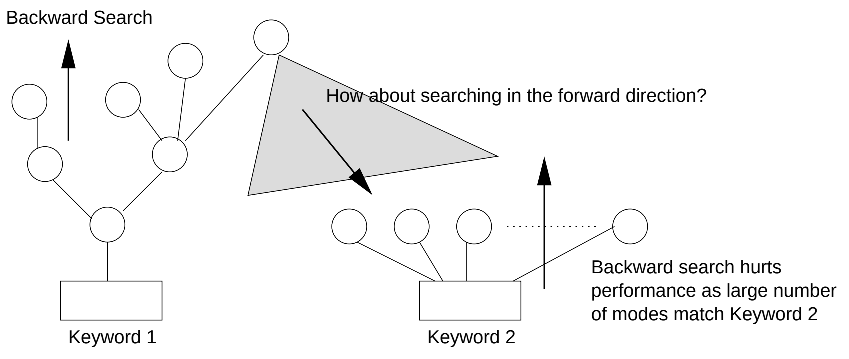

Backward Search
How about searching in the forward direction?
Backward search hurts performance as large number of modes match Keyword 2
Keyword 1
Keyword 2

Figure 2.2: Motivation for the new BANKS algorithm.

1. Since the backward algorithm associates one iterator with every keyword node, if the

number of nodes that match the keywords is high (or query contains a frequently occurring term), the algorithm would generate a large number of iterators (e.g. database in the DBLP database, John in the IMDB database) or if the keyword matches a relation name (which selects all tuples belonging to the relation).

2. The algorithm reaches a node with large fan-in: An iterator might go on to explore a large number of nodes if it hits a node with a very large fan-in (e.g. a department node in a university database which has a large number of student nodes connected to it).

Kacholia et al [KPC$^+$05] introduce a new search algorithm, which is termed *Bidirectional Search*, for schema-agnostic text search on graphs. The difference between the backward search algorithm and the bidirectional search algorithm is that unlike backward algorithm, which can only explore paths backward from keyword nodes toward the roots of answer trees, the bidirectional algorithm can explore paths forward from nodes that are potential roots of answer trees, toward other keyword nodes.

Another difference is that the bidirectional search runs only one iterator to explore backward paths from the keyword nodes as opposed the multiple iterators for the backward search algorithm. Unlike the backward search algorithm, the iterator is not a shortest path iterator since the nodes cannot be ordered to be expanded solely on the basis of the distance from the origin; the nodes are ordered by the prioritization mechanism described later (activation spread). The benefit of having a single iterator is that the amount of information to be maintained is sharply reduced.

Bidirectional search also maintains another data structure called the outgoing iterator, which expands the nodes in forward direction from potential answer roots. Every node reached by the incoming iterator is treated as a potential answer root. For each root, the outgoing iterator maintains shortest forward paths to each keyword; some of these would have been found earlier by backward search on the incoming iterator, others may be found during forward search on the outgoing iterator.

**Activation Spread.** One of the most interesting aspects of the bidirectional model is the activation spread. The bidirectional algorithm has just two iterators, so it must prioritize the nodes on some basis for execution. Thus, the paper proposes a novel prioritization scheme based on spreading activation (a kind of Pagerank which decays with distance; also refer to ObjectRank [BHP04]). This technique allows preferential expansion of paths that have less branching, and the same mechanism can be extended to implement other useful features, such as enforcing constraints using edge types to restrict search to specified search paths, or prioritizing certain paths over others.

As described in the paper, the bidirectional search algorithm can work with different ways of defining the initial activation of nodes as well as with different ways of spreading activation. The overall tree score can depend on either the edge score or the node prestige, and both need to be taken into account when defining activation to prioritize search. Nodes matching keywords are initialized with the activation content computed as:

$$a_{u,i} = \frac{nodePrestige(u)}{|S_i|}, \forall u \in S_i \tag{2.1}$$

where $S_i$ is the set of nodes that match keyword $t_i$. Thus, if the keyword node has high prestige, that node will have a higher priority for expansion. But if a keyword matches a large number of nodes, the nodes will have a lower priority. The activations from different keywords are computed separately to separate the priority contribution from each keyword. The activation spread from a node is governed by an an attenuation factor $\mu$; each node $v$ spreads a fraction $\mu$

of the received activation to its neighbors, and retains the remaining $1-\mu$ fraction. As a default the paper uses $\mu = 0.5$.

The activation from keyword $t_i$ is spread to some node $u$ in case of an incoming iterator if there is an edge $u \rightarrow v$. Amongst all such nodes, activation is divided in inverse proportion to the weight of the edge $u \rightarrow v$ (respectively, $v \rightarrow u$). This ensures that the activation priority reflects the path length from $u$ to the keyword node; trees containing nodes that are farther away are likely to have a lower score. For the outgoing iterator, activation from keyword $t_i$ is spread to some node $u$ if there is an edge $v \rightarrow u$, again divided in inverse proportion to the edge weights $v \rightarrow u$. This ensures that nodes that are closer to the potential root get higher activation, since tree scores will be worse if they include nodes that are farther away. When a node $u$ receives activation from a keyword $t_i$ from multiple edges, $a_{u,i}$ is defined as the maximum of the received activations. This reflects the fact that trees are scored by the shortest path from the root to each keyword. Other ways of combining the activation (such as adding them up) could also be used.

## 2.3 Other Systems

Given the increasing interest in search on relational databases, prior and ongoing research projects such as DBXplorer [ACD02], SphereSearch [GSW05] and ObjectRank [BHP04] have been proposed. Each of these employs a novel technique for keyword querying the databases. This section discusses the novelty of these systems and draws a critique on each of them.

### 2.3.1 DBXplorer

DBXplorer is a keyword search utility for relational databases, that has been implemented on top of the Microsoft SQL Server 2000 database server and Microsoft IIS web server. The important features of DBXplorer are:

- **Symbol Table**: It maintains a symbol table that stores mapping between keywords and database rows where it occurs. The mapping may be at a higher-level granularity if indices are available, that is, for each keyword we store the columns where the keyword occurs if an index on the column exists. Some ideas for symbol-table compaction are also discussed.
- **Database Representation**: DBXplorer represents the database as an undirected graph where nodes correspond to relations (tables) in the database and edges correspond to foreign-key links.
- **Answers**: Given a set of query keywords, DBXplorer returns all rows (either from single tables, or by joining tables connected by foreign-key joins) such that each row contains all the keywords.
- **Search Algorithm**: DBXplorer first locates all tables that contain matched keywords by looking into the symbol-table. It then enumerates all such join-trees among the tables such that the set of keywords would be contained in the join-tree. Refer Figure 2.3 where keywords are K1, K2 and K3. Black nodes correspond to tables that contain keywords. Thus, all viable join-trees containing all keywords are identied.
  The joins corresponding to each join-tree is then efficiently carried out; information stored in the symbol-table as regards row-id or the index on the column of the table is exploited while doing so. At an intuitive level, the answer-tree in BANKS can also be seen as a DBXplorer join as edges in the tree are all foreign-key references. Thus, both answer-models are similar.

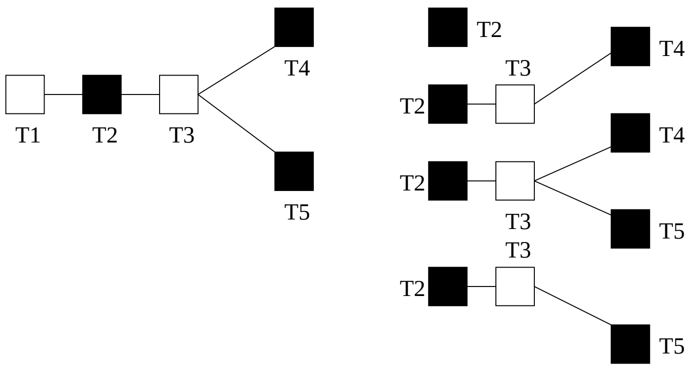


Figure 2.3: Join Trees in DBXplorer.

- **Ranking**: For ranking of answers, DBXplorer follows a simple approach. DBXplorer ranks answer-rows by the number of joins needed to form them (ties broken arbitrarily).

### Remarks

1. **Disk-based**: DBXplorer does not represent the database in memory and thus needs less memory as compared to BANKS. However, there are two issues. Firstly, if a few keywords exist in multiple tables, enumerating all the interesting join combinations could be costly. Secondly, it is not mentioned if any attempt is made to utilize the overlapping join-subtrees. If not, this could be quite costly for metadata keywords or keywords that match a lot of rows. These issues naturally do not arise in BANKS.
2. **Ranking**: DBXplorer's ranking technique is naïve. It does not identify important results from others if they are in the same table or composed of a small number of joins. For example, a meaningless answer-row with 2 papers on unrelated topics presented at the same conference is as good as an answer-row where a paper cites another. Also, there is no way to identify important tuples. So, for the query 'Pacino' on the imdb.com database, both the legendary 'Al Pacino' of 'Godfather' fame and a lesser known supporting actor 'Sal Pacino' have the same rank. This, however, is not an issue with BANKS.

### 2.3.2 ObjectRank

ObjectRank is an authority-based keyword search engine for databases. The on-the-fly tuning of the system according to the user-specified requirements gives it an edge over other systems. The distinctive features of ObjectRank are:

- **Database Representation**: Database is viewed as a labeled, directed graph. Each node 'v' corresponds to an object where object definitions have to be decided based on database schema. Each object may logically correspond to an object in real-life (like the object-oriented paradigm). For e.g., in the DBLP bibliography schema, there may be nodes corresponding to papers, authors, conferences etc. An edge is introduced whenever an object is related to another. For example, if paper P1 cites paper P2, an edge is introduced between the two as P1 $\rightarrow$ P2. Or, when author A writes paper P, an edge is introduced between the two.
- **Answers**: Answers in ObjectRank consist of whole objects. Thus, unlike BANKS or DBXplorer, no attempt is made to link objects.

- **Search Algorithm**: Each node is given global ObjectRank just like PageRank of Google, that is, based on the random-surfer model. For each keyword, there is a keyword-level ObjectRank. So, for each keyword, precompute and save object ranks of nodes obtained by setting default PageRanks of only keyword nodes to be non-zero. Object-level PageRank computation is optimized by defining cut-offs. When a user keys in keywords, these sorted object-level pageRank lists are fetched and objects in the lists are scored. Objects with the highest score are output first followed by others.
- **Ranking**: The objects in the lists fetched for each keyword are scored by the formula:

$$score_k(n) = f(GlobalObjectRank(n), KeywordObjectRank_k(n)) \quad for\ keyword\ k$$

At run time, their scores are combined :

$$score_{k1,k2,\ldots,km}(n) = score_{k1}(n) * score_{k2}(n) * \cdots * score_{km}(n) \quad for\ keywords\ k1, k2, \ldots, km$$

**Remarks**

1. **Precomputation**: Keyword-level ObjectRank values must be precomputed and stored separately for each keyword as calculation of ObjectRank at runtime is computationally expensive. However, experiments show that this precomputation is unreasonably expensive. For a database with 3 lakh nodes and 3 million edges, 74 days of precomputation is necessary on a system with moderate specifications. It should be noted that databases of such size are fairly common. For example, the DBLP database has about 3 lakh authors, 5 lakh papers and around 15 million edges.
2. **Object Defintion**: It is unclear as to how objects are defined in the database. As this is non-trivial, a safe assumption is that objects are manually defined. Defining objects and their edges is cumbersome. Also, being a manual task, it is error-prone and not a scalable solution.
3. **Answer Quality**: As mentioned, ObjectRank does not attempt to link objects as answers. This could and does result in answers with high rank that make little sense. For example, if the query is 'Database Stream Query' in the DBLP database, the user expects to see some paper related to these. However, conference nodes where papers with such titles are often published are likely to emerge as the top answers.

## 2.4 A Solution for Web Graphs

Other systems where the size of the in-memory graph can span over millions of nodes are those which do computations over web repositories. A web repository is a large special-purpose collection of web pages and associated indexes. The Stanford WebBase repository is one such repository. Efficient traversal of huge such web graphs containing several hundred million vertices and a few billion edges is a challenging problem. As a result of the missing schema structure, naive graph representation schemes can significantly increase query execution time and limit the usefulness of web repositories. In [RGM03], the authors present a novel way to structure and store web graphs so as to improve the performance of complex queries and computations over web repositories.

In [RGM03], the authors propose a novel two-level representation of graphs, called an S-Node representation. In this scheme, a graph is represented in terms of a set of smaller directed graphs, each of which encodes the interconnections within a small subset of pages. A top-level directed graph, consisting of *supernodes* and *superedges*, contains pointers to these lower level graphs. By exploiting empirically observed properties of the graphs to guide the grouping of nodes into supernodes, and using compressed encodings for the lower level directed graphs, S-Node representations provide the following two key advantages:

1. S-Node representations are highly space-efficient. Significant compression allows large graphs to be completely loaded into reasonable amounts of main memory, speeding up complex graph computations and traversals that require global/bulk access. In addition, this enables the use of simpler main-memory algorithms in place of external memory graph algorithms. Corresponding encoding techniques are described in the next subsection.

2. By representing the graph in terms of smaller directed graphs, the scheme provides a natural way to isolate and locally explore portions of the graph that are relevant to a particular query. The top-level graph serves the role of an index, allowing the relevant lower-level graphs to be quickly located.

### 2.4.1 Graph Structure

The graph structure as described in [RGM03] is as follows: let $G$ be the input graph for which a supernode graph needs to be constructed. Let $V(G)$ and $E(G)$ be the vertex set and edge set respectively, of graph $G$. The symbol $p$ refers to a page (or cluster) of the graph, or the vertex if it represents a page (or cluster). Let $P = \{N_1, N_2, ..., N_n\}$ be a partition on the vertex set of $G$. Then the graph structure consists of the following components:

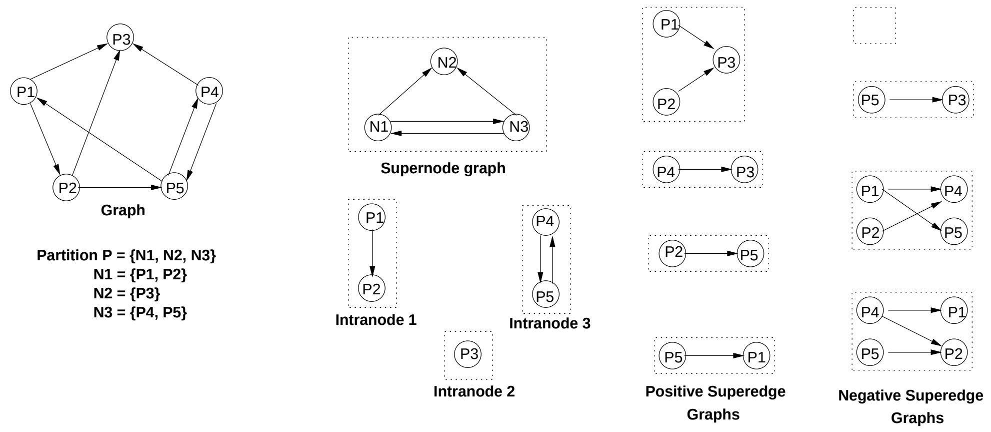


Figure 2.4: Partitioning the graph.

1. **Supernode graph.** A supernode graph contains $n$ vertices called supernodes, one for each element of the partition that are linked to each other using directed edges called superedges. Superedges are created based on the following rule: there is a directed superedge $E_{i,j}$ from $N_i$ to $N_j$ if and only if there is at least one node in $N_i$ that points to some node in $N_j$.
2. **Intranode graph.** Each partition is associated with an intranode graph, which represents all the interconnections between the nodes that belong to $N_i$.
3. **Positive superedge graph.** A positive superedge graph is a directed bipartite graph that represents all the links that point from nodes in $N_i$ to pages in $N_j$. A positive superedge graph is defined only if there is a corresponding superedge $E_{i,j}$.

4. **Negative superedge graph.** A negative superedge graph is a directed bipartite graph that represents, among all possible links that point from nodes in $N_i$ to nodes in $N_j$, those that do not exist in the actual graph.

Given a partition $P$ on the vertex set of $G$, a S-Node representation of $G$, $S(G, P)$, is a supernode graph that points to set of intranode graphs and a set of positive or negative superedge graphs. Each superedge $E_{i,j}$ points to either the corresponding positive superedge graph or the corresponding negative superedge graph, depending on which of the two superedge graphs have the smaller number of edges. The assumption is that a graph with a smaller number of edges can be encoded more compactly. While this assumption may not always be true for all compression methods, it is nevertheless useful as an approximate heuristic. This is described in more detail in the next section. The choice between positive and negative superedge graphs allows more compact encoding of both dense and sparse interconnections between nodes belonging to two different supernodes.

### 2.4.2 Desiderata

1. Nodes with similar adjacency lists should be grouped together, as much as possible. If such nodes are grouped together in the cluster, compression techniques like reference encoding can be used to achieve significant compression of intranode and superedge graphs.
2. Nodes assigned to a given cluster are connected by edges having high weights or having some lexicographic similarity. These nodes would tend to have a significant percentage of links, and thus might be traversed in a short span of time.

### 2.4.3 Clustering and other data-mining approaches

Described below an iterative process to compute a partition on the set of nodes in the graph that satisfy the desirables listed above. Let $P_0 = N_{01}, N_{02}, \ldots, N_{0n}$ be the initial coarse-grained partition. This partition is continuously refined during successive iterations, generating a sequence of partitions $P_1, P_2, \ldots, P_f$, i.e., suppose element $N_{ij}$ of partition $P_i = \{N_{i1}, N_{i2}, \ldots, N_{ik}\}$ is further partitioned into smaller sets $\{A_1, A_2, \ldots, A_m\}$ during the $i + 1^{st}$ iteration. Then, the partition for the next iteration is $P_{i+1} = \{N_{i1}, N_{i2}, \ldots, N_{i,j-1}, N_{i,j+1}, \ldots, N_{ik}\} \cup \{A_1, A_2, \ldots, A_m\}$.

The initial partition $P_0$ groups nodes based on spatial locality. In other words, all nodes in the vicinity of another based on edge weights are grouped into one element of the partition. Since the final partition $P_f$ is a refinement of $P_0$, this ensures that $P_f$ satisfies the second desideratum, i.e., all the nodes in a given element of $P_f$ are connected through strong edges.

During every iteration, one of the elements of the existing partitions is picked and further partitioned into smaller pieces. The authors claim to have tried the policy of always splitting the largest (in terms of number of nodes) element, during every iteration. However, this policy as compared with that of picking an element at random from the existing partition did not show much of a difference, i.e., the size and query performance of the S-Node representation produced by either policy was almost identical. Therefore, for the algorithm description, they assumed that an element is chosen at random. Let the element picked during the $i{+}1^{st}$ iteration be $N_{ij}$ of partition $P_i = \{N_{i1}, N_{i2}, \ldots, N_{ik}\}$. A technique known as Clustered split is then applied for splitting $N_{ij}$. Although clustered split is computationally more expensive, it is used for its performance effects on individual partitions that are smaller in size.

**Clustered Split.** Clustered Split partitions the nodes in $N_{ij}$ by using a clustering algorithm, such as k-means, to identify groups of nodes with similar adjacency lists. The output of the clustering algorithm is used to split $N_{ij}$ into smaller pieces, one per cluster. Since the authors have used this technique specifically for web graphs, other techniques could also be tried for the kind of graph that BANKS operates upon, like Correlation Clustering or K-Mediods.

To apply clustered split, a supernode graph is first built for the current partition (since it is employed repeatedly in the iteration, the authors state that it may be a good idea maintain the supernode graph throughout the refinement process, incrementally modifying the graph during every iteration). K-means clustering requires that the value of $k$, the number of clusters, be specified apriori. When applying clustered split to element $N_{ij}$, $k$ is initialized with a value equal to the out-degree of $N_{ij}$ in the supernode graph. An upper bound (which is experimentally determined based on time to convergence for run of k-means over smaller graphs) is conceived for the running time of the algorithm, and the execution is aborted if this bound is exceeded. The value of $k$ is then increased by 2 and the process repeated. If k-means repeatedly fails to converge after a fixed number of attempts, clustered split may be aborted for the current partition and algorithm proceeds to the next iteration.

**Stopping criterion.** Beginning with the initial partition $P_0$, the partition refinement using is employed using clustered split techniques, until a stopping criterion is satisfied. Ideally, the algorithm should terminate the iteration only if the current partition cannot be refined further, i.e., the clustered split technique is unable to further split any of the elements of the current partition. Since checking for this condition at every iteration is prohibitively expensive, and a stopping criterion that attempts to estimate if the "ideal stopping point" has been reached should be used.

Specifically, the refinement process may be terminated if the algorithm is forced to abort clustered split for *abortmax* consecutive iterations. Since the element to be split is chosen randomly during every iteration, this criterion can equivalently be stated as follows: iteration is stopped if, in a randomly chosen subset of the partition containing *abortmax* elements, none of the elements can be further partitioned using clustered split. For the experiments in [RGM03], *abortmax* was fixed to a fraction of the total number of elements in the partition (exact number was 6). A higher value of *abortmax* increases the accuracy of the estimate, and allows the iteration to run longer, searching for partitions that can be split further. A small value of abortmax may terminate the iteration prematurely, even if further refinements are possible.

## 2.5 Compression

We now look at the problem of how well the graphs used by BANKS can be compressed. A good compression ratio would allow for more efficient storage and transfer of graphs, and may improve the performance of the algorithm by allowing computation to be performed in faster levels of computer memory hierarchies. Good compression requires using the structural properties of the graph, but as discussed in the previous section, the kind of graphs that BANKS deals with do not belong to any special family of graphs. An example for this motivation is a graph representation that uses 20 bytes per vertex (5 ints) and 12 bytes per edge (3 floats). Total memory requirement is thus $20 \times |V| + 12 \times E$. As the database size increases, this number becomes inconvenient. For example, consider a 256MB desktop user has a database stored on his computer with a million nodes and 10 million edges. BANKS would need $20 \times 1m + 12 \times 10m = 140$MB of memory space. This is the theoretical value and any programming language explodes this further by an factor of at least 2. Thus, there is a need for reducing memory used by BANKS. This section briefly describes algorithms to efficiently compress such graphs, with an assumption that the graph structures have many shared links.

### 2.5.1 Huffman Encoding

Assuming that the indegrees and outdegrees of nodes follow a Zipfian distribution, i.e., the fraction of pages with indegree $j$ is roughly proportional to $\frac{l}{j^{\alpha}}$ for some fixed constant $\alpha$, and similarly the fraction of nodes with outdegree $j$ is roughly proportional to $\frac{l}{j^{\beta}}$ for some fixed constant $\beta$, there is a large variance in degrees. Thus it is natural to consider Huffman-based compression schemes. A simple such scheme goes through the nodes in order and lists the destination of each outedge directed from that node. Each node is assigned a Huffman codeword based on its indegree. To separate the outedges of each node a special stop symbol can be used. This approach achieves significant compression with little complexity; and it can be used in any system that wants to perform efficient computation on the compressed form of the graph.

A lot of variations of the above method have also been suggested. The compression scheme could also be based on the edges directed into each node, whichever is better. In the case where only an isomorphism of the graph needs to be stored, it may help to avoid the stop symbol. Instead, the graph can be treated as an implicit or explicit representation of the outdegree distribution, where the nodes are sorted by outdegree, and list the outedges for each node as before without the stop symbol.

### 2.5.2 Reference Encoding

Reference encoding [AM00] is a graph compression technique that is based on the following idea: if nodes $x$ and $y$ have similar adjacency lists, it is possible to compress the adjacency list of $y$ by representing it in terms of the adjacency list of $x$ (and we say that $x$ is a reference node for $y$). For example, Figure 2.5 illustrates a simple reference encoding scheme. In the figure, the adjacency lists of both $x$ and $y$ contain the entries 5, 12, 101, and 190. The adjacency list of $y$ is encoded using $x$ as a reference node. The encoded representation has two parts. The first part is a bit vector of size equal to the size of the adjacency list of $x$. A bit is set to 1 if and only if the corresponding adjacency list entry is shared between $x$ and $y$ (e.g., third bit is 1 because 12 is shared but second bit is 0 since 7 is not part of $y$'s adjacency list). The second part is a list of all the entries in $y$'s adjacency list that are not present in $x$'s list.

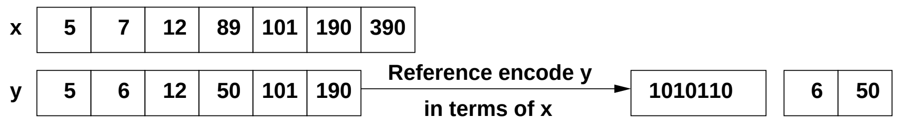


Figure 2.5: An example of Reference Encoding.

Adler and Mitzenmacher suggest an algorithm such that for a given a graph $G$, for each node $x$, it can decide whether the adjacency list for $x$ is represented as is or in terms of a reference node, and in the latter case, can also identify the particular node that will act as reference. They use the concept of an *affinity* graph $G'$, over which a directed minimum weight spanning tree can be used to generate an optimal (i.e., smallest size) reference encoded representation of $G$. Under appropriate assumptions, the running time of this algorithm is $O(nlogn)$,where $n$ is the number of nodes in the graph. For more details on the algorithm, please refer to [AM00].

As suggested in [RGM03], the supernode graph can be encoded using standard adjacency lists in conjunction with a simple Huffman-based compression scheme (each supernode is assigned a Huffman code such that those with high in-degree get smaller codes). The intranode and superedge graphs can also be encoded using the reference encoding scheme described above. In addition, wherever applicable, other easy to decode bit level compression techniques such as run length encoding (RLE) bit vectors or gap encoding adjacency lists can be employed.

These compression techniques are also applicable to the graphs generated by BANKS. However, their impact on efficiency cannot be easily determined. Thus we will not deal with them in this thesis and leave it for future work.

Chapter 3

# EMBANKS

## 3.1 Introduction

There are several challenges in designing a representation for BANKS graphs that can efficiently support a diverse range of complex queries over various schema. First, given the size and rapid improvement in disk-types, a relatively small database often has millions of tuples. A representation of these databases must efficiently store and manipulate graphs containing a few million vertices and a few billion edges. If standard data-structures are employed, only a very small portions of the graph will be able to reside in memory. As a result, complex computations and queries become highly memory intensive and time consuming. Second, BANKS graphs do not belong to any special family of graphs (e.g., trees or planar graphs) for which efficient algorithms and storage structures have been proposed in the graph clustering literature. As a result, direct adaptation of compression schemes from these domains is not possible.

In this thesis, we present EMBANKS, a framework that enables in-memory search over data graphs. The dictionary meaning of *embank* is to confine, support, or protect something with an embankment. EMBANKS is a framework that is intended to support various search models (primarily BANKS) with data structures to facilitate external memory search.

EMBANKS, in a nutshell, proceeds as follows: it first clusters the given input graph based on various parameters and stores the graph thus obtained on disk. This clustered graph is organized in such a way that the disk accesses are minimized during runtime. The search algorithm is then executed on this smaller graph to return a lot of *artificial* answers, which are then expanded to form a subgraph of the original graph. The search algorithm is then re-executed on this expanded graph to get *real* answers. This basis of the framework leads us to a series of questions - which clustering techniques should be used? What should be the size of a single cluster? How will the various weights (for both nodes and edges) in a clustered graph be computed? What is the ideal number of *artificial* answers to generate? We try to provide answers to some of these questions through the rest of this section and Section 4.

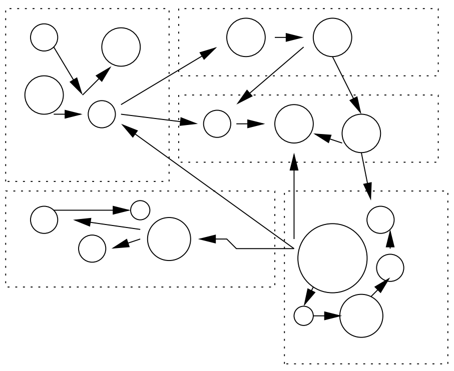


Figure 3.1: Some random clustering of nodes. Notice that area of all the rectangles is equal. Each of them represents one *page* of memory.

**Why cluster?** Clustering comes as a natural solution to the posed problem. It is an important way of exploring graphs, and has been shown to be useful in a wide variety of domains. Informally speaking, clustering is a discovery process that groups a set of nodes such that the intra-cluster similarity is maximized and the inter-cluster similarity in minimized. Clustering graphs provides a means for transforming the system into a smaller, more manageable size thereby by reducing the traversal time considerably.

We define the problem in hand as an optimization problem: Let $G = (V, E)$ be the input graph and $G' = (V', E', C_1 \ldots C_k)$ be the graph obtained after clustering. The desiderata is as follows:

1. Minimize the time taken (time for processing + number of I/O operations) to obtain and process *artificial* answers. Since $G'$ is in the compressed form, the number of *artificial* answers that should be obtained may be more than those required from the original graph.
2. $G'$ must fit in memory.
3. Minimize the total number of clusters found i.e. $k$. This prohibits the trivial solutions and eliminates trivial solutions such as a clustering with every vertex $v \in V$ in a separate cluster.

## 3.2 Answer Quality and Accuracy Measurement

Each answer tree in BANKS (and EMBANKS) has an associated node-score and edge-score:

$$\begin{aligned} nodeScore(answer) &= N = weight(rootnode) + \sum_{l \in leafNodes} weight(l) \\ edgeScore(answer) &= E = \frac{1}{1 + 1/\sum_{e \in edges} edgeWeight(e)} \end{aligned} \tag{3.1}$$

These scores are then combined into a final tree score as:

$$treeScore(answer) = EN^{\lambda} \text{ or } treeScore(answer) = \lambda N + (1 - \lambda)E \tag{3.2}$$

The results are output from output-heap sorted by their tree scores. By maintaining an upper-bound on the next best answer possible, some results are outputted from the heap before all answers are generated.

Based on this scoring model, we define *answer quality* of some answer $a$ as the difference between the score of $a$ and the score of the best answer. A good answer therefore is one whose score falls within some $\epsilon$-margin of the score of the best answer. Answers for a query $q$ having number of nodes and edges less than or equal to the maximum of number of nodes and edges of the top-10 answers obtained from BANKS is henceforth referred to as an *acceptable* answer.

## 3.3 Clustering

To build a clustered graph that efficiently supports the search and provides some guarantee on the answer quality, we need to identify a partition on the set of nodes in the graph that meet the following requirements:

- Each partition should be roughly of the same size, and this size should be a multiple of *page size*. This is crucial to achieving a reduce number of disk I/Os.

- The partition must be such that given a cluster-level subgraph, we can provide some guarantee on the answer quality for most of the queries, and ensure that the same is not compromised by a significant margin for others (i.e., the score of the answers obtained using EMBANKS is within an $\epsilon$-margin of the scores of answers obtained using BANKS).

### 3.3.1 Comparison Metric

The aim of graph compression is to get good quality results without the system occupying too much memory and taking too much time. Also, since the search algorithm of EMBANKS is independent of the compression used, it could be made a part of the metric definition. Assuming a human-interactive system, we adopt the following cluster compression metric:

Given two compressed graphs, $G_{C1}$ and $G_{C2}$, $G_{C1}$ is better than $G_{C2}$ if for a sample set of queries, both compute the best answers (say 5 best) with the constraint that the system uses less than some $m$ memory, and $G_{C1}$ results in lesser execution time than $G_{C2}$.

The proposed metric, however, is very coarse in the sense that though it is intuitive, it is hard to use in practice. The trade off between quality of results and the time required for computation imposes limitations to arrive at a theoretically optimal clustering strategy and hence we employ heuristics based on the above comparision metric.

### 3.3.2 Techniques

The problem of clustering now can be formulated in many ways. Some ways to formulate the problem and their relevance to EMBANKS have been described below.

#### Naive (Adjacency) clustering

The naive clustering technique is mostly generic clustering with certain constraints. The objective is to cluster the nodes having identical adjacency matrices, or nodes which are structurally very similar (similar adjacency lists or similar weights and prestige). It is based on the intuition that the graph size explodes when a keyword is a relation name or an attribute name, and in such cases, quite a few nodes would be similar in terms of structure, though dissimilar in content.

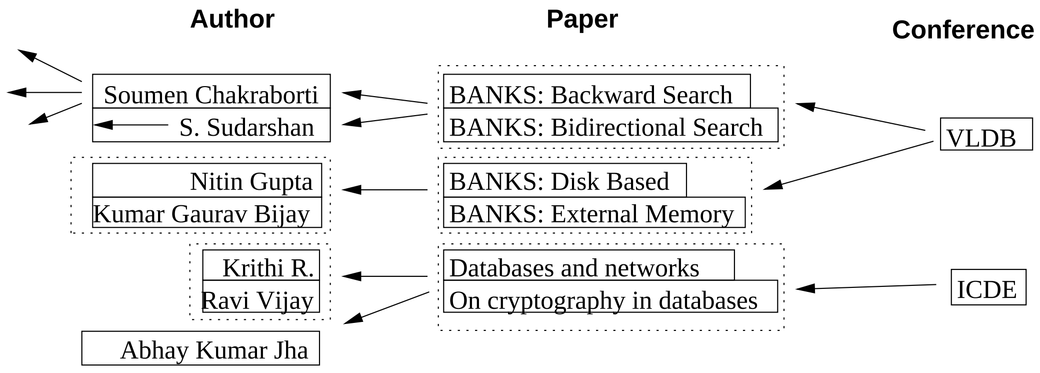


Figure 3.2: An example of Naive (Adjacency) clustering.

Though intuitively this approach seems helpful, we realized on experimentation that it fails. This was due to the fact that if the algorithm found a cluster $C$ containing a keyword node $n$, which was connected to another cluster $C'$ through the edge $(m, m') \ni m \in C, m' \in C', m]nen$, then even though the compressed graph returned an answer, a *real* answer may not exist.

**Connection clustering**

The failure of the naive-adjacency clustering technique leads to the exploration of connection-based clustering. Let $G = (V, E)$ be the input graph. Let $G' = (V', E', C_1 \ldots C_k)$ be the graph obtained after clustering. Then connection clustering is imposes that $\forall i \forall n, m \in C_i \exists n_i \ldots n_k \in C_i \ni (n_i, n_{i+1}) \in V$.

We performed experiments with connection clustering and obtained a few answers. For performance results, please refer to Table 3.1. However, naive connection clustering did not perform very well as there was no way of providing any bound on the answer score. But it did not perform very poorly either as randomization led to an equi-probable distribution of nodes into various clusters.

**Close-to-1 clustering.** A possible heuristic for the above problem is to minimize the diameter of the clusters to prevent under-weighted answers, i.e., $\forall C_r$, minimize $\max_{i,j \in C_r} sd_{i,j}$, where $sd_{i,j}$ is the shortest distance in $G$ between $v_i$ and $v_j$. This in a way means that we want to minimize the error metric for two nodes inside a cluster.

**Greedy-Minimum clustering.** Another possible heuristic is to pick up a random node to $n$ represent a cluster $C$, and add to this cluster all nodes adjacent to $n$. If there is more space in the cluster, then iteratively pick a node $n' \in C$ closest to $n$ and add all the nodes adjacent to $n'$.

## 3.3.3 Bounds on Answers

In order to output answers from the final output heap with confidence, it would be helpful to have bounds on the possible final answers that could be produced by the cluster-level graph. The clusters in an answer-tree in the clustered graph can be divided into 3 types as shown in the figure. Intermediate nodes are of interest to us.

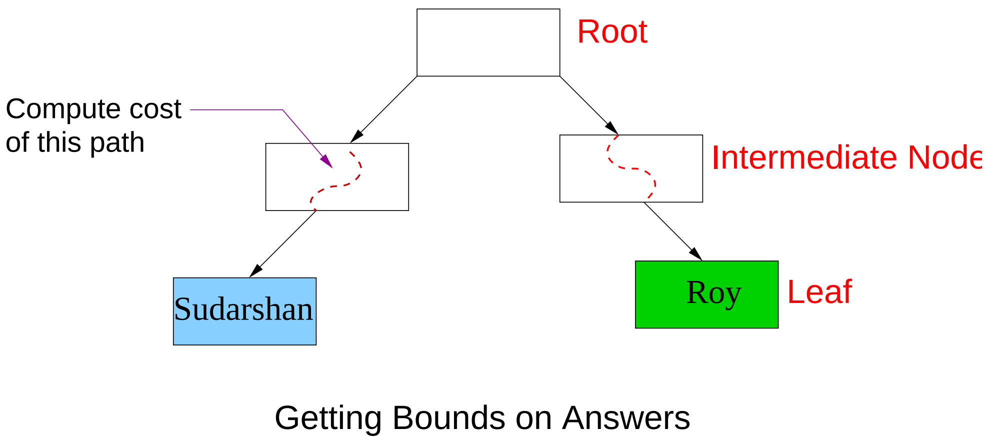


Figure 3.3: Bounds on answers

If, we maintain, for each cluster, the diameter and the cost of the cheapest path from each incoming edge to each outgoing edge (or alternatively just the overall minimum), then for an answer $A = (V, E)$ at the cluster-level, we can say that:

$$cost(BestFinalAnswer) \geq \sum_{c \in V'} cost(\text{inedge-outedge pair corresponding to c})$$

where $V' \subset V$ is the set of clusters that do not contain any keyword and

$$cost(BestFinalAnswer) \leq \sum_{e \in E} cost(e) + \sum_{c \in V} diameter(c)$$

### 3.3.4 When to get more cluster nodes?

Search happens in two phases, as discussed. However, for good performance, we need to interleave the two phases, as some good final answers might have been missed out for they need not be part of a good answer in the cluster-level graph. As a heuristic, we can get new cluster nodes when the score of an answer is lesser than that of the previous one by a certain factor $\gamma$. So, if $score(answer_{i+1}) \leq \gamma score(answer_i)$, we can expand more clusters.

### 3.3.5 Experimental Evaluation

We configured EMBANKS to experiment with bidirectional BANKS on some datasets using the above described clustering algorithms - Naive clustering, Close-to-1 clustering and Greedy-Minimum clustering. Considering the existing BANKS algorithm as the base, we calculated precision on a scale of 10 - the number of answers common between EMBANKS and BANKS, averaged over three random queries per database (DBLP: "sudarshan soumen", "nick roussopoulos christos faloutsos" and "david fernandez parametric"; IMDB: "pierce brosnan james bond", "al pacino diane keaton" and "williams carrey").

| Clustering | Precision | |
|---|---|---|
| | DBLP | IMDB |
| Naive (adjacency) | 0.0 | 0.0 |
| Naive (connection) | 4.0 (5, 3, 4) | 0.7 (2, 0, 0) |
| Close-to-1 | 4.0 (4, 2, 5) | 4.0 (10, 0, 2) |
| Greedy-minimum | 7.0 (5, 9, 7) | 3.7 (9, 2, 0) |

Table 3.1: Precision obtained using different clustering techniques

The numbers in Table 3.1 reflect an **exact** overlap. Keeping in mind the number of answers produced by BANKS, we manually categorized answers obtained from EMBANKS as acceptable or unacceptable (Refer to Table 3.2).

| Clustering | Precision | |
|---|---|---|
| | DBLP | IMDB |
| Naive (adjacency) | 0.0 | 0.0 |
| Naive (connection) | 7.0 (7, 4, 10) | 5.7 (10, 1, 6) |
| Close-to-1 | 7.0 (7, 4, 10) | 7.0 (10, 3, 8) |
| Greedy-minimum | 8.0 (7, 7, 10) | 6.0 (10, 3, 5) |

Table 3.2: Acceptable answers obtained using different clustering techniques

## 3.4 Weight Adjustment

Let $G = (V, E)$ be the input graph. Let $G' = (V', E', C_1 \ldots C_k)$ be the graph obtained after clustering, and $S = \{(u_1, v_1) \ldots (u_s, v_s)\} \subset E \ni \forall i \leq s u_s \in C_{r1}, v_s \in C_{r2}$. Then the new edge weights corresponding to $C_{r1,r2}$ and $C_{r2,r1}$ are given by:

$$\begin{aligned}
& \frac{1}{weight(C_{r1}, C_{r2})} = \frac{1}{weight(u_1, v_1)} + \cdots + \frac{1}{weight(u_s, v_s)} \\
\textbf{or} \quad & \frac{1}{weight(C_{r1}, C_{r2})} = \frac{1}{s}\left(\frac{1}{weight(u_1, v_1)} + \cdots + \frac{1}{weight(u_s, v_s)}\right) \\
\textbf{or} \quad & weight(C_{r1}, C_{r2}) = \min weight(u_i, v_i)
\end{aligned} \tag{3.3}$$

The reason for choosing the first two functions is that activation-spread is inversely proportional to the weight of the edge. So, had the nodes not been collapsed, the total activation-spread would have been the inverse sum of the inverse of all contributing edge weights. As this cluster replaces all the nodes, it must receive the whole activation that the combined nodes were receiving. Hence, $weight(C_{r1}, C_{r2})$ is given by Equation 3.3.

The node prestige of all clusters also need to be defined. Thus $\forall n_i \in C_r$

$$\begin{aligned}
& nodePrestige(C_r) = \sum_{i=1}^{s} nodePrestige(n_i) \\
\textbf{or} \quad & nodePrestige(C_r) = \max nodePrestige(n_i) \\
\textbf{or} \quad & nodePrestige(C_r) = avg\ nodePrestige(n_i)
\end{aligned} \tag{3.4}$$

We further discuss the impact of these choices in the experimental chapter.

Chapter 4

# Implementation

This chapter explains the implementation of EMBANKS. We first look at the current implementation of BANKS and suggest changes for the same. The second section describes our major contributions.

## 4.1 Current System

BANKS has been implemented in Java. The current implementation first transforms the given database into an internal graph structure. It builds an inverted keyword-index list which is stored in the database for fast lookup. The database is accessed only twice per se during a query. It is accessed once for an index lookup and then once per answer to determine what text needs to be displayed. The node prestiges, node indegrees, node outdegrees, adjacency lists, edge priorities and edge weights are stored in regular binary files for fast retrieval of bulk data. Table 4.1 shows how BANKS exactly maintains the data to save memory.

| **Array name** | **Type** | **Size** | **Function** |
|---|---|---|---|
| *nodePrestige[]* | float | $\|N\|$ | $[i]$ is the prestige of node $i$ |
| *nodeIndeg[]* | int | $\|N\|$ | $[i]$ is the indegree of node $i$ |
| *nodeOutdeg[]* | int | $\|N\|$ | $[i]$ is the outdegree of node $i$ |
| *adjacencyOffset[]* | int | $\|N\|$ | $[i]$ is the offset of node $i$'s adjacency list in $adjacentNodes[]$ |
| *adjacentNodes[]* | int | $\|E\|$ | if $e_i = (u, v)$, then $[i] = v$ |
| *edgeP[]* | float | $\|E\|$ | edge priority |
| *edgeW[]* | float | $\|E\|$ | edge weight |

Table 4.1: Various arrays that BANKS creates for graph representation.

BANKS code has four modules: default, datasource, search and util. The Datasource and Search packages work in sync to process the answers while the default package handles JDBC and HTTP protocols:

1. **Default**: The default package implements the top-level of BANKS - it handles database connections, detects tables and foreign-key references and does precomputation. The package is also responsible for the user interface.
2. **Datasource**: The datasource package is next in hierarchy. It tskes on the work from the Datasource package and constructs/loads the BANKS graph corresponding to the query.

3. **Search**: The search package encapsulates the main search algorithm. It begins by parsing the input query. All search algorithms (such as the backward expanding and bidirectional searches) are implemented within this package.
4. **Util**: The util package is at the lowest level in the hierarchy. It consists of the backbone classes needed to run BANKS. The package implements a 'Tree' class for answer trees and a 'TreeScorer' class for ranking the answers. It also implements Heaps. Java objects use a lot of memory for hashsets, vectors and similar classes. In order to alleviate this problem, BANKS relies on a package known as com.sosnoski [Sos] which implements various flavors of hashsets and hashtables like IntHashSet, IntStringHashMap.

We observed that BANKS keeps some redundant information in $nodeIndeg[]$ and $nodeOutdeg[]$ for the bidirectional search. Given the above structure, at least one of $nodeIndeg[]$ or $nodeOutdeg[]$ can be eliminated for the backward search algorithm and both for the bidirectional search, saving an additional 5-10% of memory space.

### 4.1.1 Pruning and other trivial optimizations

We did some trivial optimizations on the existing graph model in BANKS. The first of these was eliminating $nodeIndeg[]$ and (or) $nodeOutdeg[]$ for the bidirectional (resp. backward) search. Since BANKS introduces reverse edges in the bidirectional search, the indegree and outdegree of the node will always be the same. Thus, we have $\forall i$:

$$\begin{aligned} nodeIndeg[i] &= \frac{adjacencyOffset[i+1] - adjacencyOffset[i]}{2} \\ nodeOutdeg[i] &= \frac{adjacencyOffset[i+1] - adjacencyOffset[i]}{2} \end{aligned} \tag{4.1}$$

For backward search algorithm, the indegree and outdegree of a node can be different. Hence, either of the two must be maintained. Therefore, $\forall i$

$$nodeOutdeg[i] = adjacencyOffset[i+1] - adjacencyOffset[i] - nodeIndeg[i] \tag{4.2}$$

We also eliminated a lot of nodes and edges which were meant for transitivity. i.e., if a relation $R$ had only foreign-keys and primary-keys referenced by some other relation, then $R$ is in all probability a transitive relation and has no content of its own. All nodes corresponding to such relations are pruned from the graph.

## 4.2 EMBANKS

EMBANKS needs to cluster the graph nodes and also expand/read clusters from disks. Searching needs to be done on the graphs twice, which is simply implemented by calling the search function twice.

### 4.2.1 The $Cluster()$ function

The $Cluster$ function takes as input a graph and maximum cluster size and returns a representation of the clustered graph. This representation of the clustered graph is supported by three arrays: $nodeMapping[]$, $nodeOrder[]$ and $clusterOffset[]$.

- $nodeMapping[] : N \to N'$ is a mapping from a node to a cluster node.
- $nodeOrder[] : N' \to N*$ is a set-mapping function from cluster nodes to the set of nodes.

- *clusterOffset*[]: $N' \to N$ is the offset for a cluster in the *nodeOrder*[] array.

Refer to Algorithm 1 which represents close-to-1 clustering. Line 1-5 are merely initialization commands. Of these, the *nodeUsed*[] array is used to flag nodes that have already been assigned to some cluster. We begin by probing linearly for an unflagged node (line 7). Upon finding such a node, we add it to the cluster queue and traverse it's adjacency list to find another unflagged node, such that the ratio of the weights of forward and backward edge is closest to 1. We repeat this until we have filled up the cluster queue (line 11-30). At line 31, the algorithm assumes that we have formed a cluster. Now lines 33-36 create a mapping and nodeorder for this cluster. $nodeMapping[i] = j$ means that node $i$ has been assigned the cluster $j$. Similarly, *nodeOrder*[*clusterOffset*[$j$] ... *clusterOffset*[$j$]+*clusterSize*] are the nodes that belong to cluster $j$.

### 4.2.2 The DiskManager

Writing and reading clusters to/from disk is a novel idea. As we mentioned in the previous section, ideally each cluster should fit completely on one page and adjacent clusters should be on consecutive pages. Practically, page definitions and adjacent pages cannot be tracked by Java. So, we extend the constraint of a cluster fitting on one page by that of a cluster fitting on a small number of pages, which can be achieved by limiting the number of nodes and edges in a cluster. Having adjacent clusters on adjacent pages is trivially handled by issuing write instructions serially.

In our implementation, each cluster has a seperate directory, and a cluster and its adjacent cluster are consecutively numbered. Cluster writes are issued in order of their numbering, which tricks the operating-system disk management systems into allocating consecutive pages for the clusters.

We added a class *DiskManager* to the system which performs all the I/O operations related to clusters.

The *DiskManager* has the following important functions:

- *readCompressedGraph*()
- *writeCompressedGraph*()
- *readCluster*()
- *writeCluster*()

The *writeCompressedGraph*() and *writeCluster*() functions are for preprocessing only. They always occur in sequence and are preceded by a call to the *Cluster*() function and thus this whole pre-process is time consuming. The *readCompressedGraph*() function is called once every time the system is restarted and loads the cluster-level graph into memory. *readCluster*() is called once per cluster to load the clusters contained in the answers obtained in the first phase.

**Algorithm 1** Close-to-1 Clustering Algorithm

```
n=<number of nodes>
clusterSize=<size of a cluster>
nodeUsed[1 ... n] = false
clusterNumber = 0
orderPointer = 0
for i = 0 to n do
  if !nodeUsed[i] then
    queSize = 0
    queue[queSize++] = i
    nodeUsed[i] = true
    while queSize < clusterSize do
      minIndex = -1
      minValue = inf
      for j = adjacenyOffset[i] to adjacencyOffset[i+1] do
        if !nodeUsed[adjacentNode[j]]) then
          w1 = edgeW[j]
          w2 = log()
          if w1>w2 then
            ratio = w1/w2
          else
            ratio = w2/w1
          end if
          if ratio < minValue then
            minValue = ratio
            minIndex = j
          end if
        end if
      end for
      queue[queSize++] = minIndex
    end while
  end if
  clusterOffset[clusterNumber] = orderPointer
  for j = 0 to queSize do
    nodeOrder[orderPointer++] = queue[j]
    nodeMapping[queue[j]] = clusterNumber
  end for
  clusterNumber++
end for
```

Chapter 5

# Accuracy Constraints

The numbers in Table 3.2 show that EMBANKS does not produce as many good answers as BANKS does. The intuitive reason for this is that EMBANKS is not expanding all of the required clusters in the second phase, which means that none of the *artificial* answers contains these clusters. This, in turn, would probably happen either due to a poor scoring model at the cluster level or a weakness in the algorithm in the first phase or the second phase. On deeper exploration, we found a property of the bidirectional BANKS which may lead to this problem. We discuss this in the first section.

## 5.1 The Minimality Syndrome

Let $S_i$ be the set of nodes containing keyword $k_i$. Then in the directed graph model of BANKS, a response or answer to a keyword query is a minimal rooted directed tree, embedded in the data graph, and containing at least one node from each $S_i$. A resulting tree is an answer tree only if the root of the tree has more than one child. If the root of a tree $T$ has only one child, and all the keywords are present in the non-root nodes, then the tree formed by removing the root node is also present in the result set and has a higher relevance score. This characteristic is known as the **root-minimality** of directed trees. The next two subsections present two more definitions of minimality for various search algorithms.

### 5.1.1 Minimality of answers

Let $k_1 \dots k_w$ be the input keywords. Let $A = (V_A, E_A)$ be an answer such that $V_A \in V, E_A \in E$. Then $\nexists A' = (V'_A, E'_A) \ni A'$ is an answer and $A' \supset A, V' \supset V$. This effectively means that given an answer containing all the keywords, there does not exists any other answer which is a superset of this answers. Each answer thus is a *minimal Steiner tree.*

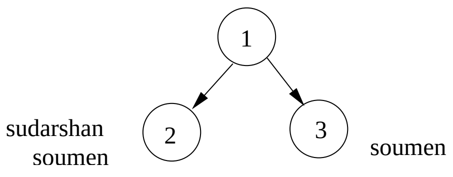


Figure 5.1: An example of a non-minimal answer.

Consider the example in Figure 5.1. For the query ”sudarshan soumen”, the two possible answers are $\{2\}$ and $\{1,2,3\}+\{(1,2),(1,3)\}$. However, only the first of these is steiner-minimal. We believe that this characteristic of the answers returned by some search systems (such as XRank) could result in the poor quality of answers produced by EMBANKS.

### 5.1.2 Minimal Intermediate Path

In order to reduce the amount of information to be maintained, bidirectional BANKS keeps on a single incoming and outgoing iterator for node exploration. Since the algorithm prioritizes nodes using factors other than distance from the keyword node, it is possible that after finding one path from some node $v$ to a keyword $t_i$, it may later find a shorter path; the distance update propagation has to be done each time a shorter path is found. This update propagation results in loss of information with respect to the longer answer - which could have been very useful in the second phase of EMBANKS.

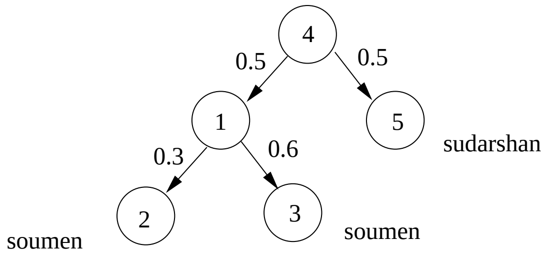


Figure 5.2: A case where the bidirectional search misses answers.

Consider the example in Figure 5.2. For the same query as above, i.e., "sudarshan soumen", the graph in consideration should ideally produce two answers. However, $N_1$ stores only the best path to keyword "soumen", i.e., the edge $(1,2)$. This is with the hope that the edge $(1,3)$ will be explored sometime during a forward expansion which may not always happen. This leads to the production of only $\{1,2,4,5\}+\{(1,2),(4,1),(4,5)\}$ and the other answer is lost.

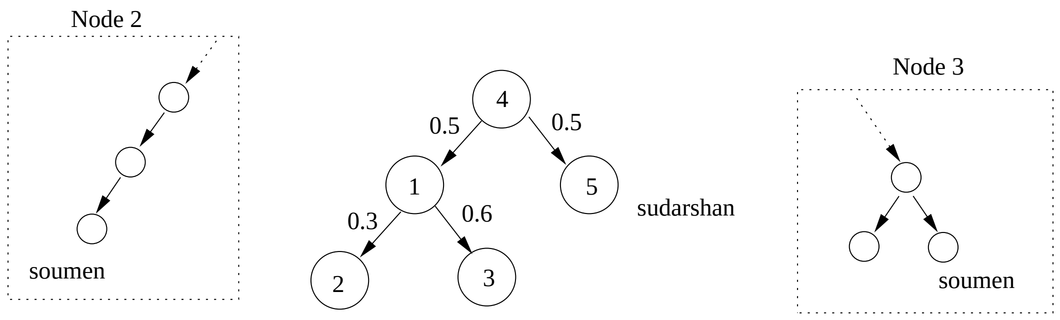


Figure 5.3: A case where the bidirectional search produces poor answers.

The seriousness of this problem is reflected in Figure 5.3. The lost answer may be a much better answer after the clusters have been expanded. Bidirectional BANKS on the other hand may not even find this cluster as a part of any of its answer, leading to a serious drop in quality. The backward BANKS algorithm provides us with a solution to this problem since it does not miss any answers. The large number of iterators and the huge task of data maintenance done by the backward search algorithm is now used to produce more diverse answers. This intern leads to the inclusion of a larger number of relevant clusters thereby increasing answer quality.

## 5.2 Utilizing that extra space

Given a user-specified amount of memory $m$ that BANKS is allowed to use, if the size of the compressed graph plus the size of the clusters expanded after the first phase is less than $m$, EMBANKS expands more clusters that it finds relevant to the given query. We expect that a greater number of nodes in the expanded graph will result in better quality of answers. A naive method to choose these extra clusters is to include all those clusters that have not been selected for expansion after the first phase and contain any of the keywords. If the total size of such clusters exceeds $m$, then we can expand any random subset of these clusters.

### 5.2.1 Introducing extra nodes in the graph

The second method for utilizing the extra memory space is to introduce more nodes to the graph, other than those obtained from the *artificial* answers. Some such nodes could be the nodes containing the keywords.

Another set of such nodes can be obtained from a parallel clustering of the nodes based on similarity measures other than proximity. A heuristic based algorithm for this kind of clustering begins by sorting keywords based on frequency. Among these keywords, those having frequency greater than some threshold value $f$ are selected. Let these keywords be $k_1, \ldots, k_n$. Then using the symbol table or disk resident indices, the tuples (also called nodes) corresponding to these keywords are located and their activation content is initialized to $a_{u,i}$ as discussed in Section 1.3. If a node has multiple keywords, the activation content is simply the sum of activations due to individual keywords. The algorithm then forms clusters in the following manner: It first selects the node having highest activation content and treats it as a cluster $c$. As a result of activation spread from this node, the activations of all nodes adjoining $c$ need to be updated. The next highest node which hasn't been alloted a cluster is then associated with $c$ if and only if the size of the cluster is less than the system page size. In case the node does not fit, a new cluster $c'$ is created and the node with the highest activation is alloted to this cluster. The process is repeated till the coarsest level of the is attained, i.e., all nodes have been associated with some cluster.

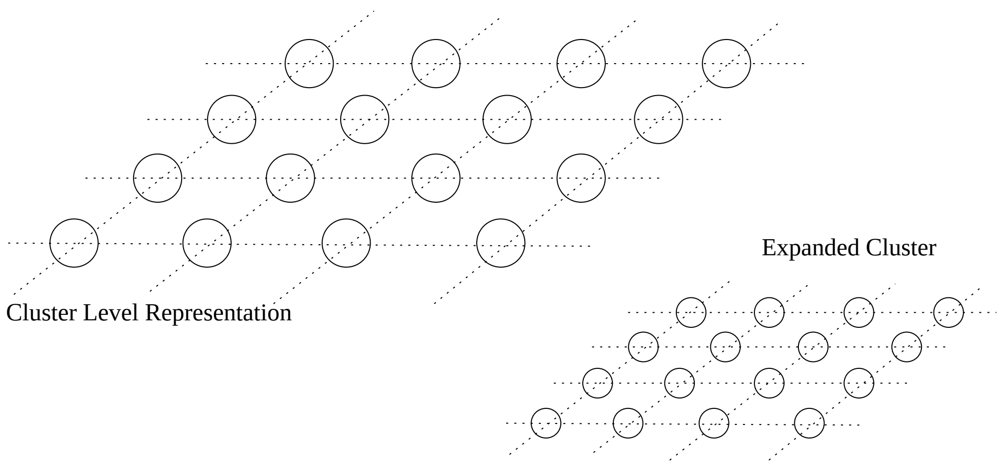


Figure 5.4: The first definition of multigranularity: a two-level graph.

We assume that at the end of above iterations, each cluster occupies not more than one page. The following properties should be maintained as the algorithm proceeds: Let $u = group(v_i, v_j)$

be a node obtained by clustering the nodes $v_i$ and $v_j$. Then:

1. The activation of the cluster node $u$ is given by $activation(u) = activation(v_i) + activation(v_j)$
2. If the nodes $v_i$ and $v_j$ are derived from the the same relation and have exactly the same adjacency lists (here same means having the same edge weights as well), the adjacency list for $u$ is same as the adjacency list for $v_i$ and the text content of node $u$ is given by $T(u) = T(v_i) \cup T(v_j)$.
3. The cluster must be tightly bound, i.e., $\forall C_i \in C, \forall v \in C_i, \frac{\sum e(|adj(v) \cap C_i|)}{|C_i|} \geq \alpha$, where $\alpha$ is some input threshold value.

An interesting feature of the above approach is that when two nodes are considered for merging, other factors such as ontological similarities can also be integrated without really affecting the algorithm as a whole. This reflects the fact that the algorithm described here may also be applicable not only to BANKS, but to other keyword query engines as well.

### 5.2.2 The second definition of Multigranularity

One more way of using the maximum available memory is to expand a few clusters, instead of introducing new clusters into the graph. The graph thus obtained will now have granularity attached to nodes instead of the graph itself. That is, there can be clusters and nodes in the same graph (ref Figure 5.5).

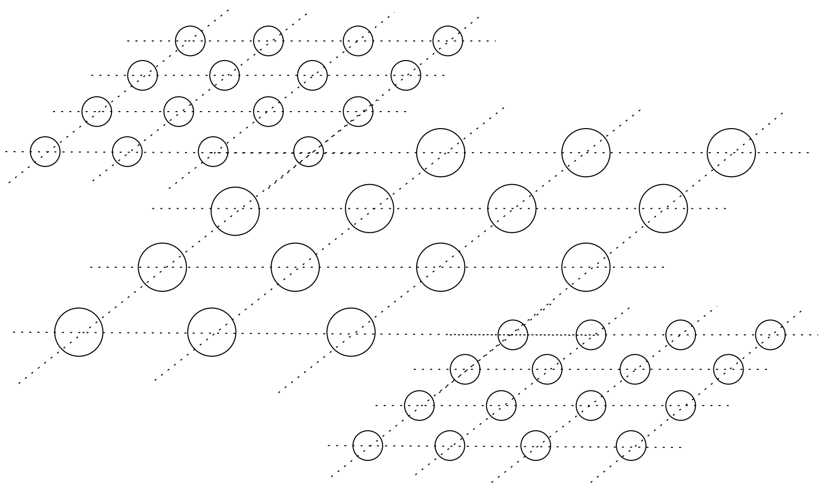


Figure 5.5: The second definition of multigranularity: an interleaved graph.

This expansion of clusters can also be done on the fly as and when some answer is generated. This will lead to an iterative expansion of the graph, which will eventually transform into a node-level graph. We expect the answer quality to increase rapidly as the graph becomes finer, since the algorithm can now eliminate a lot of poor answers immediately upon generation.

However, this model also creates a lot of complexities, the most major of them being that the scoring model now needs to be changed. A scoring model based on *indegrees* and *outdegree* of nodes will now not work since nodes and clusters are two separate entities with their own scoring model, and an ad-hoc mixture of the two will not normalize the scores in a desirable manner.

Chapter 6

# Experimental Results

We conducted several experiments to measure the performance characteristics of EMBANKS using the BANKS algorithm.

1. **Memory Usage:** *How much memory was earlier consumed by BANKS per database and how small have we been able to make it?* Keeping in spirit with the main theme of this thesis, these experiments establish the usefullness of the EMBANKS framework.
2. **Speed:** *Is there a performance improvement when EMBANKS is used?* We perform experiments to demonstrate that not only does EMBANKS reduce the initial load time by an order of magnitude, but despite the high number of disk accesses, it also outperforms both the BANKS algorithms in terms of time and the number of nodes explored.
3. **Weight Selection:** *Is harmonic mean a better measure for edge weight or minimum?* We try to answer this and analogous questions with respect to precision of the results.

## 6.1 Experimental Setup

BANKS has been implemented in Java v1.5.0 and uses PostgreSQL v7.4.2. The JAVA code connects to the the database using JDBC. Queries are posted through a servlet hosted using the Apache Tomcat web-server v5.0.28.

All the experiments were performed on dual Intel Xeon 3 GHz Processors with EM64T, 4 GB RAM, 2 X 300 GB SATA in RAID-1 hard disk running Debian Linux 2.6.14. The experiments were conducted on the following four datasets:

1. **DBLP**: The DBLP database has 4.28 lakh paper, 2.81 lakh author, 1.11 lakh cites and 9.51 lakh writes tuples. Cites nodes link citations between papers and writes nodes represent the paper-author combinations. Thus, in all, there are 17.71 lakh nodes and $21.24 \times 2 = 42.48$ lakh edges.
2. **US Patents**: The US Patent database has 1.75 lakh category, 4.14 lakh citation, 1.75 lakh coname, 11.44 lakh inventor and 5 lakh patent tuples. Inventor points to patents, patents point to category, coname points to a patent and citations link two patents. Thus, in all, there are 24.08 lakh nodes and $26.47 \times 2 = 52.94$ lakh edges.
3. **IMDB**: The IMDB (Internet Movie Database) has 0.34 lakh cite, 1.04 lakh movie, 6.44 lakh person and 9.58 role tuples. Role tuple maps to a person and a movie. A cite tuple links two movies. Thus, there are 17.40 lakh nodes and $19.84 \times 2 = 39.68$ lakh edges.
4. **IIT Bombay ETD**: The IIT Bombay Thesis database has 30 department, 505 faculty, 11 programs, 1592 students, 1811 thesis and thesis2 tuples. In all, it has 4329 nodes and $5377 \times 2 = 10754$ edges.

As discussed in Section 3.3, we have implemented three clustering ideas: Naïve-connection clustering, Close-to-1 clustering and Greedy-minimum clustering. The size of a cluster has been fixed to 100 for all the experiments. The second phase of BANKS is run after 100 answers have been returned by the first phase.

## 6.2 Memory

We consider the DBLP database. The DBLP database has 17.71 lakh nodes and 42.48 lakh edges. After clustering, we are left with 17.71 thousand nodes and 13.78 lakh edges. So, BANKS needs 49.9 MB of RAM while EMBANKS needs 18.3 MB of RAM (without keyword to cluster mapping in database) and 11.2 MB (with the mapping in database). A point to note here is that though nodes have substantially reduced, edges have reduced by a small factor only.

## 6.3 Speed

In this section, we present results from experiments to test if EMBANKS is significantly slower than BANKS. This is expected as the search algorithm runs twice for EMBANKS and the clusters have to be read from the disk. We found that clustering the graph in different ways had little impact on the speed. As the cluster-size is fixed, size of the clustered graph was roughly the same for all clusterings. Also, owing to randomization and connection-based clustering, even the naïve technique did not perform poorly in the second phase of the search when the clusters had to be expanded. We compare both the nodes touched and nodes explored to produce the first 10 answers as well as the time taken to generate the first answer. We consider three types of 2 keyword queries : queries where both keywords match many nodes, queries where both keywords match few nodes and queries where a keyword matches many nodes while the other matches few nodes.

| Type | Query | System | Time (s) | Nodes Touched | Nodes Explored |
|---|---|---|---|---|---|
| (High, High) | Database | Bidirectional | 4.56 | 274369 | 35441 |
| | Stream | EMBANKS | 6.29 | 30484 | 9056 |
| (Low, Low) | Sudarshan | Bidirectional | 0.54 | 33524 | 3948 |
| | Soumen | EMBANKS | 1.045 | 26578 | 7401 |
| (Low, High) | Nick | Bidirectional | 0.72 | 156131 | 24990 |
| | XML | EMBANKS | 1.731 | 44517 | 14072 |

Table 6.1: Speed: Bidirectional BANKS vs. EMBANKS

The above table shows that EMBANKS is slower than BANKS. However, the difference is tolerable. In the above experiments, 'Nodes Touched' and 'Nodes Explored' for EMBANKS is the sum of the nodes touched (or explored) in both the clustered and the expanded graph. *It was also observed that caching has a good impact on the speed. For example, when the query 'Database Stream' was rerun, only 0.482 sec were needed.*

## 6.4 Weights

We discussed, in Section 3.4, possible choices for Node-prestige and Edge-weight. We did some experiments with two queries on DBLP, $Q_1$: 'Database stream' (both keywords matching many nodes) and $Q_2$: 'Sudarshan Soumen' (both matching few nodes) and found the following result using soft criterion of acceptability as defined earlier.

| **Edge Weight** | **Node Prestige** | **Precision** | |
|---|---|---|---|
| | | $Q_1$ | $Q_2$ |
| Min | Sum | 2 | 7 |
| Min | Max | 0 | 7 |
| Min | Avg | 0 | 7 |
| Harmonic Mean | Sum | 2 | 6 |
| Harmonic Mean | Max | 0 | 7 |
| Harmonic Mean | Avg | 0 | 7 |
| Inverse Sum | Sum | 2 | 8 |
| Inverse Sum | Max | 0 | 7 |
| Inverse Sum | Avg | 0 | 7 |

Table 6.2: Number of relevant answers obtained with EMBANKS.

Thus, experimentally for the query-set discussed, we find that there is high correlation between the the choice of node-prestige function and the precision. We found that the combination of inverse sum for edge-weight (parallel resistance) and sum for node-prestige works best.

**Remark:** It is worth noting that the performance of EMBANKS has been bad on the 'database stream' case. This prompted us to further analyze this case. We chose few similar queries and experimented as shown in the Table 6.3.

In the following table, 'origin size' denotes the number of tuples in the database matching the corresponding keywords. Setting limits on the number of answers produced in the first phase will impact the quality of final answers. So we experimented with two different limits on the number of answers produced in the first phase: 100 and 400.

| **Query** | **Origin Size** | **Precision** | |
|---|---|---|---|
| | | **limit=100** | **limit=400** |
| database stream | (7595, 411) | 2 | 5 |
| john xml | (3218, 1450) | 8 | 8 |
| time concurrency | (12734, 1194) | 6 | 10 |
| xml query | (1450, 3236) | 10 | 10 |

Table 6.3: Experiments with keywords matching many nodes

For most queries matching many keywords, there are many good answers and we find a chunk of them. For 'database stream' however, there are only a few top-quality answers (12) and we find few of them only.

Getting more answers helps our case as more clusters are introduced. That this idea failed for other type of keywords but works for keywords with big origin size. This indicates that we could use this as a heuristic for adding more clusters.

Furthermore, experiments show that whenever we start missing answers, there is a stark difference in the tree-scores of consecutive answer-trees. This can thus be an indicator of when to fetch new clusters.

Chapter 7

# Conclusion

**Conclusion.** As the amount of information stored in databases increases, so does the need for efficient search algorithms. Keyword search enables information discovery without requiring from the user to know the schema of the database, SQL or QBE-like interface, and the roles of various entities and terms used in the query. Given the current systems, an increase in database size requires an increase in memory, which is unacceptable for simple workstations. The first part of the thesis identified this problem in BANKS and looked into other similar systems facing the same problem. We also looked at techniques suggested for an analogous system [RGM03].

The second part of the thesis introduced EMBANKS, a framework for an optimized disk-based search system, which is intended to alleviate the system of the aforementioned problem. EMBANKS has been developed with a vision to support various search models (primarily BANKS) with data structures to facilitate external memory search. As was described, EMBANKS, apart from reducing the required memory size, also sped up the database load time and run time for various queries when compared to the both the existing algorithms.

However, as reflected by the numbers in Table 3.2, EMBANKS did not produce as many good answers as BANKS did. A possible reason for this was that EMBANKS did not expand all of the required clusters in the second phase, which meant that none of the *artificial* answers contained these clusters. This, in turn, happened either due to a poor scoring model at the cluster level or a weakness in the algorithm in the first phase or the second phase. The third part of the thesis discussed properties of the bidirectional BANKS which may have led to this problem, and measures to rectify the same.

We have demonstrated that the cluster representation proposed in EMBANKS enables in-memory processing of very large database graphs. We have also established through detailed experimentation that EMBANKS can significantly reduce database load time and query execution times when compared to the original BANKS algorithms.

**Future Work.** One area that still needs to be worked on relates to compression of clusters. Compression is crucial for global/bulk access computations and mining tasks and thus may help reduce a lot of disk-accesses and lead to increasingly large cluster sizes. Questions such as how big is a cluster representation, or equivalently, how big a search graph can we represent in a given amount of main memory using some clustering scheme are still unanswered.

A lot of good answers that are generated by BANKS are not found using EMBANKS. This is mostly due to some missing clusters which were essential for an answer, but were not included in the set of clusters that are expanded. An example for this is the Naive Clustering based system, where if two keyword nodes are in the same cluster, but are not connected through a path in that cluster (or are connected through a long path in the general case), then an answer

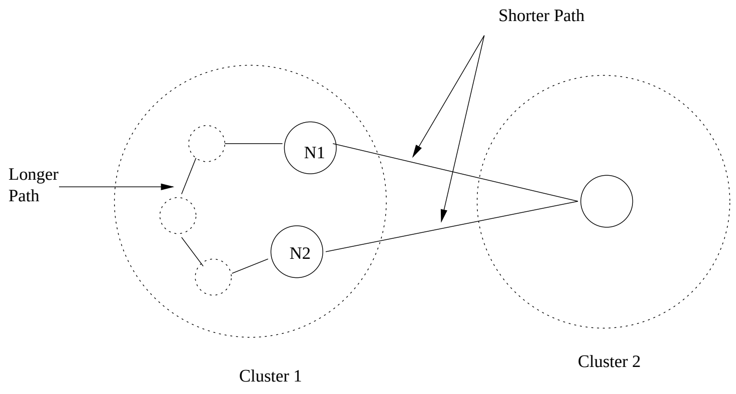


Figure 7.1: A shorter path through an external cluster.

will not be found. However, these nodes could actually be connected through a short path if some neighboring cluster had also been expanded (Refer to Figure 7.1). Thus, this inclusion of more clusters based on a given set of clusters provides a window to an area of future work.

Another potential area of work is adaptive query processing and it's application to the introduction of extra nodes. That is, given a keyword query, the algorithm should be able to determine (based on statistics computed over time) which clusters may be useful for that query. The algorithm could then also determine how many answers does it need to find at the cluster-level for expansion.

# Bibliography

[ACD02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: enabling keyword search over relational databases. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 627–627, New York, NY, USA, 2002. ACM Press.

[AM00] M. Adler and M. Mitzenmacher. Towards compressing web graphs. Technical report, IEEE, Amherst, MA, USA, 2000.

[BGVW00] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. On external memory graph traversal. In *Symposium on Discrete Algorithms*, pages 859–860, 2000.

[BHP04] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB '04: Proceedings of the 30th international conference on Very large data bases*, 2004.

[BNH$^+$02] Gaurav Bhalotia, Charuta Nakhe, Arvind Hulgeri, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[GSW05] Jens Graupmann, Ralf Schenkel, and Gerhard Weikum. The spheresearch engine for unified ranked retrieval of heterogeneous xml and web documents. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 529–540. VLDB Endowment, 2005.

[HP02] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB '02: Proceedings of the 28th international conference on Very large data bases*, 2002.

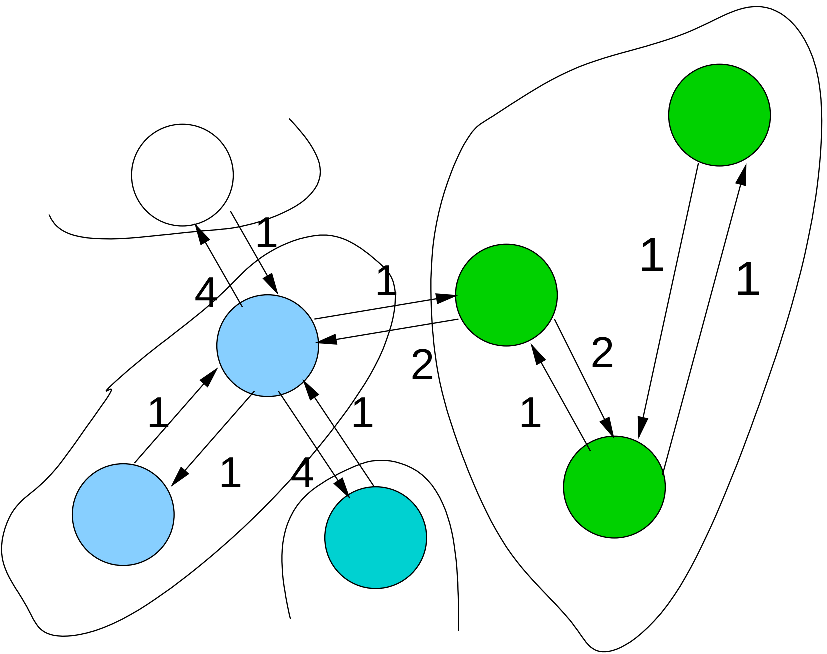
[KPC$^+$05] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 505–516. VLDB Endowment, 2005.

[LD89] P.-A. Larson and V. Deshpande. A file structure supporting traversal recursion. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 243–252, New York, NY, USA, 1989. ACM Press.

[RGM03] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *ICDE '03: Proceedings of the IEEE Intl. Conference on Data Engineering, March 2003.*, 2003.

[Sos] Dennis Sosnoski. Type-specic collections library. Technical report, Sosnoski Software Solutions, Inc. http://www.sosnoski.com/opensrc/tclib/docs/index.html.

[ZZ94] J. Leon Zhao and Ahmed Zaki. Spatial data traversal in road map databases: a graph indexing approach. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 355–362, New York, NY, USA, 1994. ACM Press.


1
4
1
2
1
1
2
1
1
1
4
1
1